МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

ОТЧЕТ

по лабораторной работе №5
по дисциплине «Алгоритмы и структуры данных»

Тема: Бинарные деревья поиска

Студент гр. 8304 Сани З. Б

Преподаватель Фирсов М.А.

Санкт-Петербург
2019

Цель работы.

Получить опыт работы с бинарными деревьями поиска, изучить на практике особенности их реализации.

Постановка задачи.

Вариант 17.

По заданным различным элементам построить АВЛ-дерево. Для построенной структуры данных проверить, входит ли в неё элемент, и если входит, то удалить элемент е из структуры данных. Предусмотреть возможность повторного выполнения с другим элементом.

Описание алгоритма.

Дерево AVL-это самобалансирующееся двоичное дерево поиска (BST), в котором разница между высотами левого и правого поддеревьев не может быть больше одного для всех узлов. Дерево AVL гарантирует логарифмический рост высоты дерева позволяющее быстро выполнять основные операции дерева поиска: добавление, удаление и поиск узла.

Когда мы удаляем узел, возникают три возможности.

1) узел, который будет удален, является листом: просто удалите из дерева.

2) узел для удаления имеет только один дочерний элемент: скопируйте дочерний элемент в узел и удалите дочерний элемент

3) узел, подлежащий удалению, имеет два дочерних элемента: найдите inorder преемника (successor) узла. Скопируйте содержимое преемника inorder на узел и удалите преемника inorder. Обратите внимание, что предшественник (predecessor) inorder также может быть использован.

AVL delete Implementation

Пусть W-узел, подлежащий удалению

1) Выполните стандартное удаление BST для W.

2) начиная с W путешествуйте вверх и найдите первый несбалансированный узел. Пусть Z-первый несбалансированный узел, Y-более высокий дочерний элемент Z, а X - более высокий дочерний элемент

Y. обратите внимание, что определения X и Y отличаются от вставки здесь.

3) повторно сбалансируйте дерево, выполнив соответствующие вращения на поддереве, коренящемся с Z. может быть 4 возможных случая, которые необходимо обработать, поскольку X, Y и Z могут быть организованы в 4 способа. Ниже приведены возможные механизмы 4:

а) Y является левым потомком Z и X является левым потомком y (слева левое дело)(Left Left Case)

б) Y является левым потомком Z и X-это право ребенка y (слева справа случае)(Left Right Case)

c) y-правильный ребенок z, а x-правильный ребенок y (правый правый случай)(Right Right Case)

д) Y-это право ребенка на Z и X является левым потомком г (справа слева случае)(Right Left Case)

После фиксации Z нам, возможно, придется также исправить предков Z.

Описание основных структур данных и функций.

Функция main осуществляет всё взаимодействие с пользователем, она приводит к созданию всех остальных используемых структур данных.

Специально для решения данной задачи был написан класс AVLtree, реализующий AVL дерево с использованием динамической памяти.

Этот класс имеет следующий интерфейс:

- insert - позволяет добавить новое значение в дерево
- delete - позволяет удалить переданное значение, если оно содержится в дереве
- searchELem - позволяет проверить наличие в дереве переданного значения,
- height - возвращает высоту дерева,
- printBracketNotation - возвращает скобочное представление дерева.

Тестирование.

Программа была успешно протестирована. Ниже приведены основные проверочные входные данные.

 - Number found and deleted

 - Sorry that Number doesn't exist in tree

AVL Tree	Elem to delete	Output	Balanced AVL
13(5(3)(8))(20(18)(22))	13		**18(5(3)(8))(20()(22))**
6(5(4))(11(10)(15(13)(16))	6		**10(5(4))(15(11()(13))(16))**
32(7(2)(24()(31)))(42(40(37))(120))	37		**32(7(2)(24()(31)))(42(40)(120))**
4(2(1)(3))(6(5)(7))	4		**5(2(1)(3))(6()(7))**
50(25(15(10)(20))(40(30)))(60(55)(80(70)))	60		**50(25(15(10)(20))(40(30)))(70(55)(80))**
32(7(2)(24()(31)))(42(40(37))(120))	9		**32(7(2)(24()(31)))(42(40(37))(120))**
17(12(9)(14))(67(50(23)(54))(76(72)))	17		**23(12(9)(14))(67(50()(54))(76(72)))**
50(25(15(10)(20))(40(30)))(60(55)(80(70)))	32		**50(25(15(10)(20))(40(30)))(60(55)(80(70)))**

```
       ::::AVL TREE APP::::
::::1 Enter Number to insert Node
::::2 Delete Number if it exists in tree --(Main Task)
::::3 Read from File
::::0 Exit

Choose Option and Click Enter: 3
> FilePath:
Your input: 5 8 3 18 13 20 22
AVL Bracket view: 13(5(3)(8))(20(18)(22))
Height of Tree: 2

:::Delete 13 if it exists
Элемент, который вы искали есть в дереве!
Deleted Succesfully 13
AVL Bracket view: 18(5(3)(8))(20()(22))
Height of Tree: 2
-----------------------------------------
Your input: 16 11 5 6 4 10 13 15
AVL Bracket view: 6(5(4))(11(10)(15(13)(16)))
Height of Tree: 3

:::Delete 6 if it exists
Элемент, который вы искали есть в дереве!
Deleted Succesfully 6
AVL Bracket view: 10(5(4))(15(11()(13))(16))
Height of Tree: 3
-----------------------------------------
Your input: 2 32 42 40 120 24 37 7 31
AVL Bracket view: 32(7(2)(24()(31)))(42(40(37))(120))
Height of Tree: 3

:::Delete 37 if it exists
Элемент, который вы искали есть в дереве!
Deleted Succesfully 37
AVL Bracket view: 32(7(2)(24()(31)))(42(40)(120))
Height of Tree: 3
-----------------------------------------
```

```
Your input: 1 2 3 4 5 6 7
AVL Bracket view: 4(2(1)(3))(6(5)(7))
Height of Tree: 2

:::Delete 4 if it exists
Элемент, который вы искали есть в дереве!
Deleted Succesfully 4
AVL Bracket view: 5(2(1)(3))(6()(7))
Height of Tree: 2
-----------------------------------------
Your input: 50 10 40 25 15 20 55 80 60 30 70
AVL Bracket view: 50(25(15(10)(20))(40(30)))(60(55)(80(70)))
Height of Tree: 3

:::Delete 60 if it exists
Элемент, который вы искали есть в дереве!
Deleted Succesfully 60
AVL Bracket view: 50(25(15(10)(20))(40(30)))(70(55)(80))
Height of Tree: 3
-----------------------------------------
Your input: 2 32 42 40 120 24 37 7 31
AVL Bracket view: 32(7(2)(24()(31)))(42(40(37))(120))
Height of Tree: 3

:::Delete 9 if it exists
Простите, но такого элемента нет
AVL Bracket view: 32(7(2)(24()(31)))(42(40(37))(120))
Height of Tree: 3
-----------------------------------------
Your input: 12 9 23 14 17 50 67 76 54 72
AVL Bracket view: 17(12(9)(14))(67(50(23)(54))(76(72)))
Height of Tree: 3

:::Delete 17 if it exists
Элемент, который вы искали есть в дереве!
Deleted Succesfully 17
AVL Bracket view: 23(12(9)(14))(67(50()(54))(76(72)))
Height of Tree: 3
-----------------------------------------
```

```
Your input: 50 10 40 25 15 20 55 80 60 30 70
AVL Bracket view: 50(25(15(10)(20))(40(30)))(60(55)(80(70)))
Height of Tree: 3

:::Delete 32 if it exists
Простите, но такого элемента нет
AVL Bracket view: 50(25(15(10)(20))(40(30)))(60(55)(80(70)))
Height of Tree: 3
----------------------------------------
```

Выводы.

В ходе работы был получен опыт работы с бинарным деревом поиска Avl в контексте данной задачи, изучены методы работы с ним. Была написана реализация бинарного дерева поиска Avl через динамическую память.

CODE

main.cpp

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <stdlib.h>
#include <vector>
#include "avl.hpp"


void ReadFromFile(AVLtree<int> &avlTree, const std::string
&filename,const std::string &test);
void insert(AVLtree<int> &avlTree,int const &n);
void deleteKey(AVLtree<int> &avlTree,int const &n);
void utility(AVLtree<int> &avlTree, std::string const &message,int
type);


void insert(AVLtree<int> &avlTree,int const &n){
      std::cout << "Inserted Succesfully " << n << std::endl;
      avlTree.insert(n);
}
void deleteKey(AVLtree<int> &avlTree,int const &n){
    bool found;
    avlTree.searchElem(n, avlTree.root,found);
    if(found){
        std::cout<<"Элемент, который вы искали есть в
дереве!"<<std::endl;
        std::cout << "Deleted Succesfully " << n << std::endl;
        avlTree.deleteKey(n);
    }else{
        std::cout<<"Простите, но такого элемента нет"<<std::endl;
    }
}

void utility(AVLtree<int> &avlTree, std::string const &message,int
type){
    do {
        std::cout<<"\n::::Enter Number to" << message << "Node (or
non-number to exit!)::::\n";
        int n;
        std::cin>>n;
```

```
        // if the user enters anything that is not integer we break
the loop
        if(std::cin.fail()){
            // helps in clearing the error flags which are set when
std::cin fails to interpret the input.
            std::cin.clear();
            // helps in clearing the stream of input in the buffer

std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
            break;
        };

        if(type == 1)
            insert(avlTree, n);
        else
            deleteKey(avlTree, n);

        avlTree.printBaracketNotation();
    } while (true);
}

std::vector<int> split(const std::string& line) {
    std::istringstream is(line);
    return std::vector<int>(std::istream_iterator<int>(is),
std::istream_iterator<int>());
}

void ReadFromFile(const std::string &initialTree,const std::string
&test)
{
    std::ifstream file(initialTree);
    std::ifstream file2(test);
    std::vector<int> deleteTest;

    if(file2.is_open()){
        std::string input;
        while (!file2.eof()) input.push_back(file2.get());
         deleteTest = split(input);
    }else{
        std::cout << "Files not opened!"<<"\n";
        return;
    }
```

```
    if (file.is_open())
    {
        int count = 0;

        while (!file.eof())
        {
            AVLtree<int> avlTree;

            std::string input;
            getline(file, input);
            std::cout<< "Your input: "<< input << "\n";
            std::vector<int> treeInput = split(input);
            for(int i : treeInput){
                   avlTree.insert(i);
            }
            avlTree.printBaracketNotation();

            std::cout<< "\n:::Delete "<< deleteTest[count] << " if
it exists\n";
            deleteKey(avlTree, deleteTest[count]);
            avlTree.printBaracketNotation();

            std::cout<<
"-----------------------------------------\n";

            count++;
        }
    }
    else
        std::cout << "Files not opened!"<<"\n";
}


int main()
{
       AVLtree<int> avlTree;
        int command;

        do{
            std::cout<<"\n\n\t\t::::AVL TREE APP::::"<<std::endl;
            std::cout<<"::::1 Enter Number to insert
Node"<<std::endl;
```

```
            std::cout<<"::::2 Delete Number if it exists in tree
--(Main Task)"<<std::endl;
            std::cout<<"::::3 Read from File"<<std::endl;
            std::cout<<"::::0 Exit"<<std::endl;

            std::cout<<"\nChoose Option and Click Enter: ";
            std::cin >> command;
            switch(command)
                    {
                        case 1:
                         {
                            std::string  message = " add ";
                            utility(avlTree, message, 1);
                            break;
                         }
                        case 2:
                           {
                                std::string  message = " delete ";
                                utility(avlTree, message, 2);
                                break;
                            }
                        case 3:
                            {
                                std::string filePath;
                                std::string filePath2;

                                std::cout << "> Tree FilePath: ";
                                std::cin >> filePath;

                                std::cout << "> Test FilePath: ";
                                std::cin >> filePath;

                                ReadFromFile(filePath,filePath2);
                                break;
                            }
                        default:
                            std::cout<<"Sorry! wrong
input"<<std::endl;
                            break;
                    }
            }while(command != 0);
```

```
        return 0;

}
```

avl.hpp

```
#ifndef avl_hpp
#define avl_hpp

#include <stdio.h>
#include <iostream>


/* AVL node */
template <class T>
class AVLnode {
    public:
        T key;
        int balance;
        AVLnode *left, *right, *parent;

        AVLnode(T k, AVLnode *p) :
                            key(k),
                            balance(0),
                            parent(p),
                            left(NULL), right(NULL) {}
        ~AVLnode() {
            delete left;
            delete right;
        }

};

/* AVL tree */
template <class T>
class AVLtree {
    public:
        AVLtree() = default;
        AVLtree(const AVLtree<T> &) = delete;
        ~AVLtree() {
            delete root;
        }
        AVLtree &operator=(const AVLtree<T> &) = delete;
```

```
        AVLnode<T> *root = nullptr;
        bool insert(T key);
        void deleteKey(const T key);
        void searchElem(T key,AVLnode<T> *root, bool &found);
        void bracketNotation(AVLnode<T>* root, std::string& str);
        void printBaracketNotation();
        int height(AVLnode<T> *n);

    private:
        AVLnode<T>* rotateLeft(AVLnode<T> *a);
        AVLnode<T>* rotateRight(AVLnode<T> *a);
        AVLnode<T>* rotateLeftThenRight(AVLnode<T> *n);
        AVLnode<T>* rotateRightThenLeft(AVLnode<T> *n);
        void rebalance(AVLnode<T> *n);
        void setBalance(AVLnode<T> *n);
};

#endif /* avl_hpp */

/* AVL class definition */

template <class T>
int AVLtree<T>::height(AVLnode<T> *n) {
    if (n == NULL)
        return -1;
    return 1 + std::max(height(n->left), height(n->right));
}

template <class T>
void AVLtree<T>::setBalance(AVLnode<T> *n) {
    n->balance = height(n->right) - height(n->left);
}


template <class T>
void AVLtree<T>::rebalance(AVLnode<T> *n) {
    setBalance(n);

    if (n->balance == -2) {
        if (height(n->left->left) >= height(n->left->right))
            n = rotateRight(n);
        else
```

```
            n = rotateLeftThenRight(n);
        }
        else if (n->balance == 2) {
            if (height(n->right->right) >= height(n->right->left))
                n = rotateLeft(n);
            else
                n = rotateRightThenLeft(n);
        }

        if (n->parent != NULL) {
            rebalance(n->parent);
        }
        else {
            root = n;
        }
    }

    template <class T>
    bool AVLtree<T>::insert(T key) {
        if (root == NULL) {
            root = new AVLnode<T>(key, NULL);
        }
        else {
            AVLnode<T>
                *n = root,
                *parent;

            while (true) {
                if (n->key == key)
                    return false;

                parent = n;

                bool goLeft = key < n->key;
                n = goLeft ? n->left : n->right;

                if (n == NULL) {
                    if (goLeft) {
                        parent->left = new AVLnode<T>(key, parent);
                    }
                    else {
                        parent->right = new AVLnode<T>(key, parent);
                    }
```

```
                rebalance(parent);
                break;
            }
        }
    }

    return true;
}

template <class T>
void AVLtree<T>::deleteKey(const T delKey) {
    if (root == NULL)
        return;

    AVLnode<T>
        *n       = root,
        *parent  = root,
        *delNode = NULL,
        *child   = root;

    while (child != NULL) {
        parent = n;
        n = child;
        child = delKey >= n->key ? n->right : n->left;
        if (delKey == n->key)
            delNode = n;
    }

    if (delNode != NULL) {
        delNode->key = n->key;

        child = n->left != NULL ? n->left : n->right;

        if (root->key == delKey) {
            root = child;
        }
        else {
            if (parent->left == n) {
                parent->left = child;
            }
            else {
                parent->right = child;
```

```
            }

            rebalance(parent);
        }
    }
}


template <class T>
AVLnode<T>* AVLtree<T>::rotateLeft(AVLnode<T> *a) {
    AVLnode<T> *b = a->right;
    b->parent = a->parent;
    a->right = b->left;

    if (a->right != NULL)
        a->right->parent = a;

    b->left = a;
    a->parent = b;

    if (b->parent != NULL) {
        if (b->parent->right == a) {
            b->parent->right = b;
        }
        else {
            b->parent->left = b;
        }
    }

    setBalance(a);
    setBalance(b);
    return b;
}

template <class T>
AVLnode<T>* AVLtree<T>::rotateRight(AVLnode<T> *a) {
    AVLnode<T> *b = a->left;
    b->parent = a->parent;
    a->left = b->right;

    if (a->left != NULL)
        a->left->parent = a;
```

```
    b->right = a;
    a->parent = b;

    if (b->parent != NULL) {
        if (b->parent->right == a) {
            b->parent->right = b;
        }
        else {
            b->parent->left = b;
        }
    }

    setBalance(a);
    setBalance(b);
    return b;
}

template <class T>
AVLnode<T>* AVLtree<T>::rotateLeftThenRight(AVLnode<T> *n) {
    n->left = rotateLeft(n->left);
    return rotateRight(n);
}

template <class T>
AVLnode<T>* AVLtree<T>::rotateRightThenLeft(AVLnode<T> *n) {
    n->right = rotateRight(n->right);
    return rotateLeft(n);
}

template <class T>
void AVLtree<T>::searchElem(T key,AVLnode<T> *root, bool &found){
      if (root==NULL)
           found = false;
       else if (key < root->key)
           searchElem(key,root->left,found);
       else if (key>root->key)
           searchElem(key,root->right,found);
       else
           found = true;
}


template <class T>
```

```
void AVLtree<T>::bracketNotation(AVLnode<T>* root, std::string&
str){
     if (root == NULL)
         return;

     std::string s = std::to_string(root->key);
     for(char i : s){
         str.push_back(i);
      }

     if (!root->left && !root->right)
         return;

     // for left subtree
     str.push_back('(');
     bracketNotation(root->left, str);
     str.push_back(')');

     // only if right child is present to
     // avoid extra parenthesis
     if (root->right) {
         str.push_back('(');
         bracketNotation(root->right, str);
         str.push_back(')');
     }
 }

template <class T>
void AVLtree<T>::printBaracketNotation(){
     std::string brack;
     int h = height(root);
     bracketNotation(root, brack);
     std::cout << "AVL Bracket view: " << brack<< std::endl;
     std::cout << "Height of Tree: " << h<< std::endl;

 }
```